**8086 Microprocessor – Simple Explanation of Internal Architecture**

The **Intel 8086** is a **16-bit** microprocessor introduced in 1978, widely used in early computers. It processes instructions, performs calculations, and communicates with memory and input/output devices. To make it work efficiently, it is internally divided into two main sections:

1. **Bus Interface Unit (BIU)**
2. **Execution Unit (EU)**

These units work together to process and execute instructions.

**1. Bus Interface Unit (BIU)**

**What is BIU?**

The **Bus Interface Unit (BIU)** is responsible for interacting with memory and input/output devices. It handles the fetching of instructions and data from memory and communicates through buses.

**Main Components of BIU:**

**(a) Instruction Queue**

* The **8086** uses a **six-byte instruction queue**.
* It stores instructions **before the CPU needs them** (this process is called **prefetching**).
* Prefetching improves performance by reducing waiting time.

**(b) Segment Registers**

* The **8086** uses **segmented memory**, meaning memory is divided into blocks called **segments**.
* Each segment is identified by a **segment register**:
  + **CS (Code Segment):** Stores the address of program instructions.
  + **DS (Data Segment):** Stores data used by the program.
  + **SS (Stack Segment):** Holds temporary data storage for functions and procedures.
  + **ES (Extra Segment):** Additional data storage.

**(c) Address Generation**

* The BIU generates **physical addresses** by combining:
  + A **segment address** from a segment register.
  + An **offset address** (instruction pointer or other registers).
* This allows **1MB of memory addressing** in the **8086** microprocessor.

**2. Execution Unit (EU)**

**What is EU?**

The **Execution Unit (EU)** takes the fetched instructions and executes them. It performs arithmetic, logical, and control operations.

**Main Components of EU:**

**(a) Arithmetic Logic Unit (ALU)**

* The **ALU (Arithmetic Logic Unit)** is responsible for performing:
  + **Arithmetic operations** (Addition, Subtraction, Multiplication, Division).
  + **Logical operations** (AND, OR, XOR, NOT).
  + **Comparison operations** (Equal to, Greater than, Less than).

**(b) General-Purpose Registers**

* These registers temporarily store data, addresses, and results.
* **Types of General-Purpose Registers:**
  + **AX (Accumulator Register):** Stores primary arithmetic and logic results.
  + **BX (Base Register):** Holds base memory addresses.
  + **CX (Counter Register):** Used in loop operations.
  + **DX (Data Register):** Holds additional data during calculations.
  + **SI (Source Index) & DI (Destination Index):** Used in string operations.
  + **BP (Base Pointer):** Helps in stack memory access.
  + **SP (Stack Pointer):** Keeps track of the top of the stack.

**(c) Flags Register**

* The **Flags Register** holds **status flags** that indicate the results of an operation.
* Important Flags:
  + **Zero Flag (ZF):** Set if the result is zero.
  + **Carry Flag (CF):** Set if there is an overflow in addition/subtraction.
  + **Overflow Flag (OF):** Set if the result exceeds the maximum value.
  + **Sign Flag (SF):** Determines if the result is negative.
  + **Parity Flag (PF):** Checks if the number of 1s in the result is even or odd.

**(d) Control Unit (CU)**

* The **Control Unit** acts as a **traffic manager**, controlling the flow of instructions and data.
* It ensures the **correct sequence of operations** in the processor.

**3. Instruction Pointer (IP)**

* The **Instruction Pointer (IP)** is a **16-bit register** that **stores the address of the next instruction** to be executed.
* After each instruction, the IP automatically moves to the next instruction in memory.

**4. Address Generation Unit (AGU)**

* The **AGU (Address Generation Unit)** calculates memory addresses required for **fetching data and instructions**.
* It uses:
  + **Segment registers** (to hold base addresses).
  + **Offset values** (from other registers).

**5. Control Unit (CU)**

* The **Control Unit (CU)** ensures that all parts of the processor **work together smoothly**.
* It **coordinates** execution, manages instruction flow, and generates **control signals**.
* It also controls communication with **external devices** using control bus signals.

**6. Buses in 8086**

The **8086** uses three main buses to communicate with memory and other devices:

1. **Address Bus:**
   * Carries the **memory address** from the processor to memory.
   * It is **20-bits wide**, allowing the **8086 to access up to 1MB of memory**.
2. **Data Bus:**
   * Transfers **data** between the processor and memory or I/O devices.
   * It is **16-bits wide**, allowing **16-bit data transfer** at a time.
3. **Control Bus:**
   * Carries **control signals** that tell memory and other devices what the processor wants to do.
   * Examples of control signals:
     + **Read (RD) and Write (WR) Signals** – Tell memory if data should be read or written.
     + **Interrupt Signals (INTR, INTA)** – Used for handling external events.

**Summary of 8086 Architecture**

The **8086 microprocessor** has two main sections:

* **Bus Interface Unit (BIU):** Fetches instructions, manages memory, and prefetches instructions for efficiency.
* **Execution Unit (EU):** Executes instructions, performs calculations, and processes data.

**Other Important Parts:**

* **Instruction Pointer (IP):** Keeps track of the next instruction.
* **Address Generation Unit (AGU):** Calculates memory addresses.
* **Control Unit (CU):** Manages the entire processor operation.
* **Buses (Address, Data, and Control Buses):** Help in communication with memory and input/output devices.

# **Execution Unit (EU) of 8086 Microprocessor – Detailed Explanation**

The **Execution Unit (EU)** is one of the two main components of the **8086 microprocessor**, responsible for executing instructions that have been fetched from memory. It performs arithmetic, logic, and data operations using various registers and processing units.

## ****1. Registers in the Execution Unit (EU)****

The EU contains several registers that store data, addresses, and status flags. These registers play a crucial role in processing and execution.

### ****(a) General-Purpose Registers****

* The **8086** has **eight 16-bit general-purpose registers**, which can also be used as **two separate 8-bit registers** (high and low bytes).
* These registers **store data, addresses, and intermediate results** during execution.

| **Register** | **Full 16-bit Name** | **8-bit Parts** | **Purpose** |
| --- | --- | --- | --- |
| **AX** | Accumulator Register | AH, AL | Used in arithmetic and I/O operations |
| **BX** | Base Register | BH, BL | Holds base addresses for memory access |
| **CX** | Count Register | CH, CL | Used in loops and counting operations |
| **DX** | Data Register | DH, DL | Stores extra data for operations like multiplication and division |
| **SI** | Source Index Register | - | Used in string operations as a pointer |
| **DI** | Destination Index Register | - | Used in string operations for destination |
| **BP** | Base Pointer | - | Used for accessing stack memory |
| **SP** | Stack Pointer | - | Holds the top address of the stack |

* **AX (Accumulator):** Main register used for arithmetic and logic operations.
* **BX (Base):** Often holds base addresses for memory operations.
* **CX (Count):** Commonly used as a counter in loops and shift operations.
* **DX (Data):** Used for special cases like **multiplication**, **division**, and **I/O operations**.
* **SI (Source Index) & DI (Destination Index):** Used in **string operations** and memory addressing.
* **BP (Base Pointer) & SP (Stack Pointer):** Used to manage stack memory.

### ****(b) Segment Registers****

* **8086 uses a segmented memory model**, where memory is divided into four **16-bit** segments, each addressed by a **segment register**.
* These registers store the **starting addresses** of the different memory segments.

| **Segment Register** | **Function** |
| --- | --- |
| **CS (Code Segment)** | Holds the base address of the program code |
| **DS (Data Segment)** | Stores the base address of data in memory |
| **SS (Stack Segment)** | Keeps the base address of the stack |
| **ES (Extra Segment)** | Used for additional memory access |

* The **physical address** is calculated using these segment registers along with an **offset value**.

**Formula:**

Physical Address=(Segment Register×16)+Offset\text{Physical Address} = (\text{Segment Register} \times 16) + \text{Offset}Physical Address=(Segment Register×16)+Offset

For example, if **CS = 2000H** and **IP = 1234H**, then:

Physical Address=(2000H×10H)+1234H=21234H\text{Physical Address} = (2000H \times 10H) + 1234H = 21234HPhysical Address=(2000H×10H)+1234H=21234H

### ****(c) Flags Register (Status Register)****

* The **Flags Register** contains **status flags** that reflect the result of operations and **control flags** for conditional execution.
* It is a **16-bit register**, but only **9 flags** are used in the **8086 microprocessor**.

#### ****Important Flags in the 8086 Microprocessor:****

| **Flag** | **Description** |
| --- | --- |
| **Zero Flag (ZF)** | Set if the result of an operation is **zero** |
| **Sign Flag (SF)** | Set if the result is **negative** (most significant bit = 1) |
| **Carry Flag (CF)** | Set if there is a **carry-out** from the highest bit in arithmetic operations |
| **Overflow Flag (OF)** | Set if a signed arithmetic operation **overflows** beyond its range |
| **Parity Flag (PF)** | Set if the number of 1s in the result is **even** |
| **Auxiliary Carry Flag (AF)** | Set if there is a carry from **bit 3 to bit 4** (used in BCD operations) |
| **Trap Flag (TF)** | Enables **single-step execution** for debugging |
| **Interrupt Flag (IF)** | If set, **interrupts are enabled**; if cleared, **interrupts are disabled** |
| **Direction Flag (DF)** | Determines the direction of **string operations** (increment/decrement) |

* **Flags are used in decision-making operations** like **jumps, loops, and conditional execution**.

## ****2. Working of the Execution Unit (EU)****

The **EU is responsible for executing instructions fetched by the Bus Interface Unit (BIU)**. It follows a **step-by-step execution process**:

### ****(a) Instruction Decoding****

* The **BIU fetches the instruction** from memory and places it in the **instruction queue**.
* The EU **decodes** the **opcode** to determine the operation and operands.

### ****(b) Arithmetic and Logical Operations****

* The **Arithmetic Logic Unit (ALU)** performs **mathematical** and **logical** operations.
* Example operations:
  + **Arithmetic:** ADD, SUB, MUL, DIV
  + **Logical:** AND, OR, XOR, NOT
  + **Comparison:** CMP (Compare two values)

**Example:**

assembly

CopyEdit

MOV AX, 05H ; Load AX with 5

ADD AX, 03H ; Add 3 to AX (AX = 8)

The **Zero Flag (ZF)** will be **0** because the result is **not zero**.

### ****(c) Data Movement****

* The **EU moves data** between registers, memory, and I/O devices.
* Example operations:
  + **MOV:** Transfer data from one register/memory to another.
  + **PUSH/POP:** Store or retrieve data from the stack.
  + **IN/OUT:** Input and output operations.

**Example:**

assembly

CopyEdit

MOV BX, 2000H ; Load BX with 2000H

MOV AX, [BX] ; Move data from memory (address 2000H) to AX

### ****(d) Control Flow Management****

* The EU **controls program execution** using jump and branching instructions.
* Important operations:
  + **JMP (Jump):** Directly move to a new instruction address.
  + **CALL/RET:** Used for function calls.
  + **LOOP:** Executes a block of code multiple times.

**Example:**

assembly

CopyEdit

MOV CX, 05H ; Set counter to 5

LOOP\_LABEL:

DEC CX ; Decrement CX

JNZ LOOP\_LABEL ; Jump back if CX is not zero

* The **JNZ (Jump if Not Zero)** checks the **Zero Flag (ZF)** to decide whether to continue looping.

### ****(e) Flags Management****

* The **EU updates the Flags Register** based on the result of operations.
* Conditional instructions use these flags to **control program flow**.

**Example:**

assembly

CopyEdit

CMP AX, BX ; Compare AX and BX

JZ EQUAL ; Jump if Zero Flag (ZF) is set (AX = BX)

## ****3. EU Interaction with Other Units****

* **BIU (Bus Interface Unit):** The EU **requests data/instructions** from the BIU.
* **AGU (Address Generation Unit):** Helps in calculating memory addresses.
* **Clock Synchronization:** The EU executes **instructions in multiple machine cycles** based on complexity.

## ****Summary of Execution Unit (EU)****

| **Component** | **Function** |
| --- | --- |
| **General-Purpose Registers** | Store data, addresses, and results |
| **Segment Registers** | Hold base addresses of different memory segments |
| **Flags Register** | Stores the status of operations and controls branching |
| **ALU (Arithmetic Logic Unit)** | Performs arithmetic and logical operations |
| **Instruction Decoder** | Decodes fetched instructions |
| **Control Unit** | Manages execution flow and operations |

The **Execution Unit (EU)** is the **core processing unit** of the 8086, responsible for executing instructions, managing registers, and updating flags for decision-making. 🚀

# **Bus Interface Unit (BIU) of 8086 Microprocessor – Explained Simply**

The **Bus Interface Unit (BIU)** is like the **communication manager** of the **8086 microprocessor**. It connects the **CPU to memory and input/output devices**, fetching instructions and moving data between the processor and the external system.

Let’s break it down step by step in simple terms.

## ****1. Registers in the BIU****

Registers are **small storage areas** inside the CPU that hold important values. The BIU has special registers to handle memory access and instruction fetching.

### ****(a) Segment Registers****

The **8086 microprocessor uses segmented memory** to access large amounts of data.  
The BIU has **four 16-bit segment registers**, each holding the **starting address of a specific memory segment**:

| **Segment Register** | **Purpose** |
| --- | --- |
| **CS (Code Segment)** | Holds the base address of the program code. |
| **DS (Data Segment)** | Holds the base address of data (variables). |
| **SS (Stack Segment)** | Holds the base address of the stack (temporary storage). |
| **ES (Extra Segment)** | Used for extra memory operations. |

💡 **Example:**

* If the **code segment** is stored at address **2000H**, then **CS** will store **2000H**.
* If a program instruction is located **at offset 0030H**, the **physical address** will be calculated as: Physical Address=(CS×16)+Offset\text{Physical Address} = (\text{CS} \times 16) + \text{Offset}Physical Address=(CS×16)+Offset =(2000H×10H)+0030H=20030H= (2000H \times 10H) + 0030H = 20030H=(2000H×10H)+0030H=20030H This means the instruction is actually stored at memory location **20030H**.

### ****(b) Instruction Pointer (IP)****

* The **IP register** keeps track of the **next instruction** to be executed.
* After each instruction fetch, the **IP automatically increases**, so the next instruction is executed in order.

💡 **Example:**

* If **IP = 0030H**, the next instruction is at **20030H** (using CS = 2000H).
* After fetching the instruction, **IP = 0031H**, pointing to the next instruction.

### ****(c) Control Registers****

* The BIU has **control registers** that help manage memory access and control signals.
* These registers help the processor **read/write data**, manage **interrupts**, and coordinate system operations.

## ****2. How the Bus Interface Unit (BIU) Works****

The **BIU performs four main tasks** to keep the processor running smoothly:

### ****(a) Fetching Instructions****

* The **BIU fetches program instructions** from memory so the **Execution Unit (EU)** can execute them.
* It uses **CS (Code Segment) and IP (Instruction Pointer)** to calculate the **physical memory address** of instructions.
* Once fetched, the instructions are stored in a **queue** (like a waiting line).

💡 **Example:**  
If a program is stored in memory at **address 20030H**, the **BIU fetches the instruction from there** and stores it in the queue.

### ****(b) Prefetch Queue (Instruction Buffering)****

* The BIU **prefetches up to 6 bytes of instructions** from memory and stores them in a queue.
* This speeds up execution because the EU doesn’t have to wait for each instruction to be fetched.
* When the EU needs the next instruction, it simply **takes it from the queue instead of waiting** for memory access.

💡 **Think of it like this:**

* Imagine you are **reading a book aloud**, but instead of stopping to read each word, you **look ahead and read a few words in advance**.
* This makes reading **smoother and faster**, just like **prefetching speeds up processing**.

### ****(c) Transferring Data Between CPU and Memory/I/O Devices****

* The BIU also **handles data transfer** between the CPU and:
  + **Memory** (for storing and retrieving data)
  + **Input/Output devices** (like keyboards, displays, and external devices)
* It uses the **segment registers (DS, SS, ES) and offsets** to calculate physical memory addresses for data.

💡 **Example:**  
If a program wants to read a number stored at **memory address 3050H**, the **BIU fetches it** and gives it to the EU for processing.

### ****(d) Address Generation****

* The **BIU calculates the physical address** whenever the CPU needs to access memory.
* It uses the **Address Generation Unit (AGU)** for this purpose.
* The AGU combines **segment addresses** from segment registers and **offsets** to generate the **final memory address**.

💡 **Example:**  
To access data at offset **0050H** in the **data segment (DS = 4000H)**, the physical address is:

Physical Address=(4000H×10H)+0050H=40050H\text{Physical Address} = (4000H \times 10H) + 0050H = 40050HPhysical Address=(4000H×10H)+0050H=40050H

### ****(e) Generating Control Signals****

* The **BIU sends control signals** to coordinate memory operations, interrupts, and communication with external devices.
* It decides **when to read/write memory**, handle **interrupts**, and control **bus usage**.

💡 **Example of Control Signals:**

| **Signal** | **Purpose** |
| --- | --- |
| **Memory Read (MR)** | Reads data from memory. |
| **Memory Write (MW)** | Writes data to memory. |
| **I/O Read (IOR)** | Reads data from an input device. |
| **I/O Write (IOW)** | Sends data to an output device. |

### ****(f) Communicating with the External System****

* The **BIU connects the CPU with memory and external devices** using three main buses:

| **Bus** | **Function** |
| --- | --- |
| **Address Bus** | Sends memory addresses. |
| **Data Bus** | Transfers actual data. |
| **Control Bus** | Sends control signals (read, write, interrupt). |

💡 **Example of Bus Communication:**

* When you **type on a keyboard**, the BIU **receives the key input from an I/O port**, processes it, and **sends it to the CPU** for further action.
* When a program **saves a file**, the BIU **sends data to memory storage** using the buses.

## ****3. Summary of the Bus Interface Unit (BIU)****

| **Function** | **What It Does** |
| --- | --- |
| **Registers** | Stores addresses for memory access. |
| **Instruction Fetch** | Fetches program instructions from memory. |
| **Prefetch Queue** | Stores up to 6 instructions in advance for faster execution. |
| **Data Transfer** | Moves data between CPU, memory, and I/O devices. |
| **Address Generation** | Calculates physical memory addresses. |
| **Control Signals** | Manages read/write operations and external communication. |

### ****Flag Register of 8086 and Function of Each Flag – Simple Explanation****

The **flag register** in the **Intel 8086 microprocessor** is a **16-bit register** that stores various **status flags** and **control flags**. These flags help the processor make **decisions** based on the results of arithmetic and logical operations.

## ****1. What is the Flag Register?****

* The **flag register** is a **special register** that holds **important information** about the result of the last operation.
* It contains **two types of flags**:
  1. **Status Flags** – Indicate the result of operations.
  2. **Control Flags** – Control the CPU’s functioning.
* The CPU checks these flags before deciding what to do next, especially for **conditional jumps** and other decision-making operations.

## ****2. Flags in the 8086 Flag Register****

| **Flag Name** | **Bit Position** | **Purpose** |
| --- | --- | --- |
| **Zero Flag (ZF)** | 6 | Set if result = 0 |
| **Sign Flag (SF)** | 7 | Set if result is negative |
| **Carry Flag (CF)** | 0 | Set if a carry occurs |
| **Overflow Flag (OF)** | 11 | Set if signed overflow occurs |
| **Auxiliary Carry Flag (AF)** | 4 | Set if a carry occurs in BCD arithmetic |
| **Parity Flag (PF)** | 2 | Set if the number of 1s in the result is even |

Now, let's go through **each flag and its function** in detail.

## ****3. Status Flags (Indicate the Result of Operations)****

### ****1. Zero Flag (ZF) – Bit 6****

* The **Zero Flag (ZF)** is set if the result of an arithmetic or logical operation is **zero**.
* If the result is **zero**, ZF = **1**. Otherwise, ZF = **0**.
* Used in **conditional jumps** (e.g., if ZF = 1, the CPU can decide to take a certain action).

💡 **Example:**

csharp

CopyEdit

5 - 5 = 0 → ZF = 1 (because result is zero)

7 - 3 = 4 → ZF = 0 (because result is not zero)

### ****2. Sign Flag (SF) – Bit 7****

* The **Sign Flag (SF)** reflects the **sign of the result** of an arithmetic operation.
* If the result is **negative**, SF = **1**.
* If the result is **positive**, SF = **0**.
* The CPU **copies the most significant bit (MSB) of the result into SF**.

💡 **Example:**

yaml

CopyEdit

0111 1111 (+127) → SF = 0 (Positive)

1000 0001 (-127) → SF = 1 (Negative)

### ****3. Carry Flag (CF) – Bit 0****

* The **Carry Flag (CF)** is set when there is a **carry or borrow** in an arithmetic operation.
* This flag is important for **multi-byte arithmetic operations**.

💡 **Example:**

yaml

CopyEdit

1111 1111 (255)

+ 0000 0001 (1)

----------------

0000 0000 (0) with Carry = 1

Since the result exceeds **8 bits**, CF = **1** (carry occurred).

### ****4. Overflow Flag (OF) – Bit 11****

* The **Overflow Flag (OF)** is set when the result of a **signed** arithmetic operation is too large for the register.
* This means the result cannot be correctly represented in the available bits.

💡 **Example (8-bit numbers):**

yaml

CopyEdit

0111 1111 (+127)

+ 0000 0001 (+1)

----------------

1000 0000 (-128) → Overflow! → OF = 1

* The result **should** be **128**, but since 8086 uses **2’s complement**, it becomes **-128**, causing an **overflow**.

### ****5. Auxiliary Carry Flag (AF) – Bit 4****

* The **Auxiliary Carry Flag (AF)** is set if there is a **carry from bit 3 to bit 4**.
* It is used mainly in **BCD (Binary Coded Decimal) arithmetic**.

💡 **Example:**

yaml

CopyEdit

0000 1001 (9)

+ 0000 0001 (1)

----------------

0001 0000 (10) → Carry from bit 3 to 4 → AF = 1

AF is **important for BCD arithmetic**, but not commonly used in regular operations.

### ****6. Parity Flag (PF) – Bit 2****

* The **Parity Flag (PF)** is set if the number of **1’s in the result** is **even**.
* If the number of **1’s is odd**, PF = **0**.

💡 **Example:**

ini

CopyEdit

Result = 1010 1100 (4 ones) → Even → PF = 1

Result = 1010 1101 (5 ones) → Odd → PF = 0

PF is useful for **error detection** in data transmission.

## ****4. Why Are These Flags Important?****

* These flags help in **decision-making** within the CPU.
* Many **conditional branch instructions** rely on the **status of these flags** to determine execution flow.
* Examples of **instructions that use flags**:
  + JZ (Jump if Zero) – Uses the **Zero Flag**.
  + JS (Jump if Sign) – Uses the **Sign Flag**.
  + JC (Jump if Carry) – Uses the **Carry Flag**.
  + JO (Jump if Overflow) – Uses the **Overflow Flag**.
  + JP (Jump if Parity) – Uses the **Parity Flag**.

## ****5. Summary of Flags****

| **Flag Name** | **Bit** | **Purpose** |
| --- | --- | --- |
| **Zero (ZF)** | 6 | Set if result is zero |
| **Sign (SF)** | 7 | Set if result is negative |
| **Carry (CF)** | 0 | Set if a carry occurs |
| **Overflow (OF)** | 11 | Set if signed overflow occurs |
| **Auxiliary Carry (AF)** | 4 | Set if carry occurs from bit 3 to bit 4 (BCD arithmetic) |
| **Parity (PF)** | 2 | Set if the number of 1s in the result is even |

## ****Control Flags (Control Processor Behavior)****

| **Flag** | **Full Form** | **Bit Position** | **Purpose** |
| --- | --- | --- | --- |
| **TF** | Trap Flag | 8 | Enables step-by-step execution for debugging. |
| **IF** | Interrupt Flag | 9 | Enables or disables external interrupts. |
| **DF** | Direction Flag | 10 | Controls string operations (increment or decrement). |

### ****(a) Trap Flag (TF) – Bit 8****

* If **TF = 1**, the CPU executes instructions **one at a time**, useful for **debugging**.
* If **TF = 0**, normal execution continues.

### ****(b) Interrupt Flag (IF) – Bit 9****

* If **IF = 1**, the CPU allows **external interrupts** (e.g., keyboard input).
* If **IF = 0**, interrupts are **disabled**.

### ****(c) Direction Flag (DF) – Bit 10****

* **DF controls string operations** (used for data movement).
* If **DF = 0**, the string is processed **from left to right** (increment).
* If **DF = 1**, the string is processed **from right to left** (decrement).

💡 **Example:**

* Copying **HELLO** from memory:
  + **DF = 0** → Copy **H → E → L → L → O**
  + **DF = 1** → Copy **O → L → L → E → H**

## ****5. Summary of All Flags****

| **Flag** | **Bit** | **Purpose** |
| --- | --- | --- |
| **Carry (CF)** | 0 | Shows carry or borrow in arithmetic. |
| **Parity (PF)** | 2 | Checks even/odd parity of the result. |
| **Auxiliary Carry (AF)** | 4 | Used in BCD arithmetic (carry from bit 3 to 4). |
| **Zero (ZF)** | 6 | Set if result = 0. |
| **Sign (SF)** | 7 | Set if result is negative. |
| **Overflow (OF)** | 11 | Set if result is too big for register. |
| **Trap (TF)** | 8 | Enables single-step debugging. |
| **Interrupt (IF)** | 9 | Enables or disables external interrupts. |
| **Direction (DF)** | 10 | Controls string operations (left-to-right or right-to-left). |

### ****(Simple Explanation)****

The **Intel 8086 microprocessor** uses a **20-bit address bus**, which means it can access **1 MB (2^20 = 1,048,576 bytes) of memory**. However, since the registers in the 8086 are only **16-bit**, the processor cannot directly generate a 20-bit address. To solve this problem, **segmentation** is used.

### ****1. What is Segmentation in 8086?****

Segmentation is a method used in the **8086 microprocessor** to divide memory into smaller, **manageable 64 KB segments**. Instead of handling a large **1 MB address space** directly, the processor **divides memory into segments** and accesses them using a **segment address and an offset**.

In simple terms, think of memory as a **book** where:

* Each **segment** is like a **chapter** (a large block of memory).
* The **offset** is the **page number** within that chapter.
* By combining the **chapter number (segment)** and **page number (offset)**, we find the **exact location** (physical address).

### ****2. Registers Used for Addressing****

To store the **segment address**, the **8086 processor** has **four segment registers**:

| **Segment Register** | **Purpose** |
| --- | --- |
| **CS (Code Segment)** | Holds the starting address of program instructions |
| **DS (Data Segment)** | Holds the starting address of data variables |
| **SS (Stack Segment)** | Holds the starting address of the stack |
| **ES (Extra Segment)** | Used for additional memory storage |

Each of these registers contains a **16-bit segment address**. To locate a specific memory address inside a segment, a **16-bit offset** is added.

### ****3. How is a 20-bit Physical Address Generated?****

Since the **segment register** contains only a **16-bit value**, it needs to be converted into a **20-bit physical address**. This is done using a **simple formula**:

Physical Address=(Segment Address×16)+Offset Address\text{Physical Address} = (\text{Segment Address} \times 16) + \text{Offset Address}Physical Address=(Segment Address×16)+Offset Address

OR

Physical Address=(Segment Address≪4)+Offset Address\text{Physical Address} = (\text{Segment Address} \ll 4) + \text{Offset Address}Physical Address=(Segment Address≪4)+Offset Address

🔹 **What happens in this calculation?**

* The **segment address** is **shifted left by 4 bits** (same as multiplying by 16).
* The **offset address** is **added** to this shifted value to get the **final 20-bit physical address**.

### ****4. Example of Physical Address Calculation****

Let’s say we have the following values:

* **Segment Address** = 1234H
* **Offset Address** = 5678H

#### ****Step 1: Convert Segment Address to 20-bit Format****

First, we **shift the segment address left by 4 bits** (multiply by 16).

Segment Address=1234H=0001001000110100\text{Segment Address} = 1234H = 0001 0010 0011 0100Segment Address=1234H=0001001000110100 After Left Shift=00010010001101000000\text{After Left Shift} = 0001 0010 0011 0100 0000After Left Shift=00010010001101000000

Now, the segment address has become **12340H**.

#### ****Step 2: Add the Offset Address****

Now, we **add the offset address (5678H)** to the shifted segment address:

Physical Address=12340H+5678H\text{Physical Address} = 12340H + 5678HPhysical Address=12340H+5678H =179B8H= 179B8H=179B8H

So, the **final 20-bit physical address** is **179B8H**.

### ****5. Maximum Addressable Memory (1 MB Limit)****

Since the **physical address** is **20-bit**, the maximum possible address is:

FFFF0H+FFFFH=FFFFFH=1,048,575 bytes(1MB)\text{FFFF0H} + \text{FFFFH} = \text{FFFFFH} = 1,048,575 \text{ bytes} (1 MB)FFFF0H+FFFFH=FFFFFH=1,048,575 bytes(1MB)

So, the **Intel 8086 can access up to 1 MB of memory**.

### ****6. Why is Segmentation Used?****

✅ **Allows accessing more memory** than a 16-bit register can hold (64 KB limit).  
✅ **Efficient memory management**, making it easy to organize different parts of a program.  
✅ Helps in **modular programming**, where **code, data, and stack are stored in separate memory segments**.

### ****7. Limitations of Segmentation****

❌ **Limited to 64 KB per segment**, making programming complex.  
❌ **Overlapping segments** can create memory fragmentation.  
❌ **Difficult address calculations**, as programmers must manage segments and offsets manually.

To overcome these issues, later processors (like the **80386**) introduced a **flat memory model**, eliminating segmentation.

### ****8. Summary****

* **8086 uses segmentation** to manage **1 MB memory** with **16-bit registers**.
* **Physical address = (Segment × 16) + Offset**.
* **Memory is divided into 64 KB segments** (CS, DS, SS, ES).
* **Segmentation improves memory access but has limitations**.

### ****Special Functions of Registers in Intel 8086 (Other Than Data Storage) – Simple Explanation****

The **8086 microprocessor** has several general-purpose registers (**AX, BX, CX, DX, SP**), which store data during operations. However, each register also has a **special function** beyond just holding data. Let’s look at their special roles in detail.

### ****1. AX Register (Accumulator Register) – Special Role in Arithmetic and I/O Operations****

✅ The **AX register** is the most important register for arithmetic and I/O operations.

* **Used in Arithmetic Operations:**
  + When the CPU performs **multiplication or division**, the AX register **automatically** stores the result.
  + Example: If you multiply two numbers, the result is stored in **AX (or AX + DX for larger results).**
* **Used in Input/Output (I/O) Operations:**
  + The **AX register** is often used for **data transfer** between the CPU and external devices.
  + Example:

assembly

CopyEdit

MOV DX, 03F8H ; Port address (e.g., serial port)

IN AX, DX ; Read data from port 03F8H into AX

🔹 **Why is this special?**  
The **AX register** is **automatically used** for many CPU operations, so programmers don’t always need to specify it.

### ****2. BX Register (Base Register) – Special Role in Memory Addressing****

✅ The **BX register** is **commonly used as a base address for memory operations**.

* **Used in Indexed Addressing Mode:**
  + The BX register can store a **memory address**, which allows the CPU to access data efficiently.
  + Example: Accessing an array of numbers stored in memory.

assembly

CopyEdit

MOV BX, 2000H ; Load base address of the array

MOV AL, [BX] ; Load the first byte from memory at address 2000H

* **Used with SI and DI Registers for Complex Addressing:**
  + **BX + SI (Source Index) or BX + DI (Destination Index)** is commonly used for complex memory operations like **moving blocks of data**.

🔹 **Why is this special?**  
Instead of using a fixed memory address, the BX register allows **dynamic** memory access, making programs more efficient.

### ****3. CX Register (Count Register) – Special Role in Loop and Repetitive Operations****

✅ The **CX register** is mainly used as a **counter** for loops and repetitions.

* **Used in Loop Instructions:**
  + The CX register **automatically decrements** in a loop until it reaches **zero**.
  + Example: A simple loop that repeats 5 times.

assembly

CopyEdit

MOV CX, 5 ; Set loop counter to 5

LOOP\_LABEL:

; Some operations

LOOP LOOP\_LABEL ; Decrease CX and repeat until CX = 0

* **Used in String and Shift Operations:**
  + In **string operations**, the CX register holds the length of the string.
  + In **bit shift operations**, it determines the number of shifts.

assembly

CopyEdit

MOV CX, 3

SHL AX, CL ; Shift AX left by 3 bits (CX value)

🔹 **Why is this special?**  
The **CX register is automatically used** in loops, so programmers don’t need to manually check and update counters.

### ****4. DX Register (Data Register) – Special Role in I/O and Large Arithmetic Operations****

✅ The **DX register** is important for **port I/O operations and large arithmetic calculations**.

* **Used in Input/Output (I/O) Operations:**
  + The **DX register** holds the **port address** when reading or writing to hardware devices.
  + Example: Sending data to a printer or serial port.

assembly

CopyEdit

MOV DX, 03F8H ; Load port address (COM1)

OUT DX, AL ; Send data in AL to the port

* **Used as an Extension for Large Multiplication and Division:**
  + The DX register works with **AX for 32-bit calculations**.
  + Example: Multiplying two 16-bit numbers to get a 32-bit result.

assembly

CopyEdit

MOV AX, 1234H

MOV DX, 0000H ; Clear DX before multiplication

MUL BX ; DX:AX = AX × BX (result stored in DX and AX)

🔹 **Why is this special?**  
The **DX register is automatically used** in specific operations, especially **port I/O and large arithmetic operations**.

### ****5. SP Register (Stack Pointer) – Special Role in Stack Management****

✅ The **SP (Stack Pointer) register** is used to manage the **stack**, a special memory area for storing temporary data.

* **Tracks the Top of the Stack:**
  + When **data is pushed onto the stack**, SP **decreases**.
  + When **data is popped from the stack**, SP **increases**.
  + Example:

assembly

CopyEdit

PUSH AX ; Store AX on the stack (SP decreases)

POP AX ; Retrieve AX from the stack (SP increases)

* **Used for Function Calls and Interrupts:**
  + When a **function (subroutine) is called**, the CPU **automatically** pushes the return address onto the stack.
  + When an **interrupt occurs**, the CPU **automatically** saves registers onto the stack.

🔹 **Why is this special?**  
The **SP register is critical** for program execution because it **automatically** manages function calls, returns, and interrupts.

### ****Final Summary****

| **Register** | **Special Function (Other Than Data Storage)** |
| --- | --- |
| **AX (Accumulator)** | Used in arithmetic and I/O operations |
| **BX (Base Register)** | Used as a base address for memory access |
| **CX (Count Register)** | Used as a loop counter in repetitive tasks |
| **DX (Data Register)** | Used in port I/O operations and large arithmetic calculations |
| **SP (Stack Pointer)** | Manages the stack for temporary storage and function calls |

### ****Conclusion****

Each of these **general-purpose registers** has a **special function**, which helps the **8086 microprocessor** work efficiently. These features allow the CPU to perform **arithmetic calculations, memory access, loops, I/O operations, and stack management** without extra programming effort.

### ****1-1 Program to Read a Character from User and Display on Screen****

assembly

CopyEdit

MOV AH, 01H ; Read character

INT 21H ; Store in AL

MOV DL, AL ; Move to DL for display

MOV AH, 02H ; Display character

INT 21H

INT 20H ; Exit

### ****1-2 Program to Read a Character Without Echo****

assembly

CopyEdit

MOV AH, 08H ; Read character without echo

INT 21H

MOV DL, AL

MOV AH, 02H ; Display character

INT 21H

INT 20H

### ****1-3 Program to Read a Character and Display on a New Line****

assembly

CopyEdit

MOV AH, 01H

INT 21H

MOV DL, 0DH ; Carriage Return

MOV AH, 02H

INT 21H

MOV DL, 0AH ; Line Feed

INT 21H

MOV DL, AL ; Print character

INT 21H

INT 20H

### ****1-4 Accept Small Letter and Display Capital****

assembly

CopyEdit

MOV AH, 01H

INT 21H

SUB AL, 20H ; Convert to uppercase

MOV DL, AL

MOV AH, 02H

INT 21H

INT 20H

### ****2-1 Add Two 8-bit Numbers****

assembly

CopyEdit

MOV AL, 05H ; First number

ADD AL, 03H ; Second number

MOV DL, AL

MOV AH, 02H

INT 21H

INT 20H

### ****2-2 Subtract Two 8-bit Numbers****

assembly

CopyEdit

MOV AL, 08H

SUB AL, 03H

MOV DL, AL

MOV AH, 02H

INT 21H

INT 20H

### ****2-3 Multiply Two 8-bit Numbers****

assembly

CopyEdit

MOV AL, 04H

MOV BL, 03H

MUL BL

MOV DL, AL

MOV AH, 02H

INT 21H

INT 20H

### ****2-4 Divide Two 8-bit Numbers****

assembly

CopyEdit

MOV AL, 08H

MOV BL, 02H

DIV BL

MOV DL, AL

MOV AH, 02H

INT 21H

INT 20H

### ****3-1 Add Two 16-bit Numbers****

assembly

CopyEdit

MOV AX, 1000H

ADD AX, 2000H

INT 20H

### ****3-2 Subtract Two 16-bit Numbers****

assembly

CopyEdit

MOV AX, 3000H

SUB AX, 1000H

INT 20H

### ****3-3 Multiply Two 16-bit Numbers****

assembly

CopyEdit

MOV AX, 1234H

MOV BX, 0002H

MUL BX

INT 20H

### ****3-4 Divide Two 16-bit Numbers****

assembly

CopyEdit

MOV AX, 1000H

MOV BX, 0004H

DIV BX

INT 20H

### ****4 Access Marks, Calculate Percentage & Display Grade****

assembly

CopyEdit

MOV AL, 80H ; Example marks

ADD AL, 70H

ADD AL, 90H

ADD AL, 85H

ADD AL, 75H

MOV BL, 05H ; Divide by 5

DIV BL

CMP AL, 60H

JGE PASS

MOV DX, 'F'

JMP PRINT

PASS: MOV DX, 'P'

PRINT: MOV AH, 02H

INT 21H

INT 20H

### ****5-1 Block Transfer****

assembly

CopyEdit

MOV SI, 1000H

MOV DI, 2000H

MOV CX, 10

REP MOVSB

INT 20H

### ****5-2 Block Interchange****

assembly

CopyEdit

MOV SI, 1000H

MOV DI, 2000H

MOV CX, 10

REP XCHG BYTE PTR [SI], [DI]

INT 20H

### ****5-3 Move String With & Without Instructions****

assembly

CopyEdit

MOV SI, 1000H

MOV DI, 2000H

MOV CX, 10

REP MOVSB ; With instruction

INT 20H

assembly

CopyEdit

MOV SI, 1000H

MOV DI, 2000H

MOV CX, 10

CLD

LOOP: MOV AL, [SI]

MOV [DI], AL

INC SI

INC DI

LOOP LOOP

INT 20H

### ****5-4 String Reversal****

assembly

CopyEdit

MOV SI, OFFSET STRING

MOV DI, OFFSET STRING + LENGTH - 1

MOV CX, LENGTH / 2

REV\_LOOP: MOV AL, [SI]

XCHG AL, [DI]

MOV [SI], AL

INC SI

DEC DI

LOOP REV\_LOOP

INT 20H

### ****5-5 Check String Palindrome****

assembly

CopyEdit

MOV SI, OFFSET STRING

MOV DI, OFFSET STRING + LENGTH - 1

MOV CX, LENGTH / 2

PAL\_LOOP: MOV AL, [SI]

CMP AL, [DI]

JNE NOT\_PAL

INC SI

DEC DI

LOOP PAL\_LOOP

MOV DX, 'Y'

JMP PRINT

NOT\_PAL: MOV DX, 'N'

PRINT: MOV AH, 02H

INT 21H

INT 20H

### ****5-6 String Concatenation****

assembly

CopyEdit

MOV SI, OFFSET STR1

MOV DI, OFFSET STR2

CONC\_LOOP: MOV AL, [SI]

CMP AL, 0

JE COPY

INC SI

JMP CONC\_LOOP

COPY: MOV AL, [DI]

MOV [SI], AL

INC SI

INC DI

CMP AL, 0

JNE COPY

INT 20H

### ****5-7 Compare Two Strings****

assembly

CopyEdit

MOV SI, OFFSET STR1

MOV DI, OFFSET STR2

MOV CX, LENGTH

CMP\_LOOP: MOV AL, [SI]

CMP AL, [DI]

JNE NOT\_EQUAL

INC SI

INC DI

LOOP CMP\_LOOP

MOV DX, 'Y'

JMP PRINT

NOT\_EQUAL: MOV DX, 'N'

PRINT: MOV AH, 02H

INT 21H

INT 20H

### ****6-1 Display Name at Center of Screen****

assembly

CopyEdit

MOV AH, 02H

MOV DH, 12 ; Row 12

MOV DL, 20 ; Column 20

INT 10H

MOV DX, OFFSET NAME

MOV AH, 09H

INT 21H

INT 20H

### ****7-1 HEX to BCD Conversion****

assembly

CopyEdit

MOV AL, 5AH

MOV AH, 00H

DAA

INT 20H

### ****7-2 BCD to HEX Conversion****

assembly

CopyEdit

MOV AL, 29H

MOV AH, 00H

AAA

INT 20H

### ****8 Count Zeros, Positive, Negative in an Array****

assembly

CopyEdit

MOV CX, LENGTH

MOV SI, OFFSET ARRAY

ZERO\_COUNT: MOV AL, [SI]

CMP AL, 00H

JE INC\_ZERO

CMP AL, 80H

JL INC\_POS

INC\_NEG: INC NEG\_CNT

JMP NEXT

INC\_POS: INC POS\_CNT

JMP NEXT

INC\_ZERO: INC ZERO\_CNT

NEXT: INC SI

LOOP ZERO\_COUNT

INT 20H

### ****9-1 Find Largest Number in Array****

assembly

CopyEdit

MOV SI, OFFSET ARRAY

MOV AL, [SI]

MOV CX, LENGTH

FIND\_LARGEST: INC SI

CMP AL, [SI]

JGE SKIP

MOV AL, [SI]

SKIP: LOOP FIND\_LARGEST

INT 20H

### ****9-2 Find Smallest Number in Array****

assembly

CopyEdit

MOV SI, OFFSET ARRAY

MOV AL, [SI]

MOV CX, LENGTH

FIND\_SMALLEST: INC SI

CMP AL, [SI]

JLE SKIP

MOV AL, [SI]

SKIP: LOOP FIND\_SMALLEST

INT 20H